A Comprehensive Guide to Intermediate React

Section 1: Synchronizing with External Systems: The useEffect Hook

The useEffect Hook is a fundamental concept in intermediate React development. It allows components to perform side effects, which are operations that interact with the world outside of the component's render-only logic. This includes data fetching, setting up subscriptions, or manually changing the DOM.

1.1. The Philosophy of Effects: Thinking Beyond Component Lifecycles

A common pitfall for developers transitioning from class-based components is to view useEffect as a direct replacement for the componentDidMount, componentDidUpdate, and componentWillUnmount lifecycle methods. This mental model is not only incorrect but is the source of many common bugs and misunderstandings.

An Effect's lifecycle is distinct from a component's lifecycle. A component may mount, update, or unmount. An Effect, by contrast, only does two things: it *starts synchronizing* something, and later, it *stops synchronizing* it. This cycle of starting and stopping can, and often does, happen multiple times while the component is mounted.

The correct mental model is to consider useEffect as an "escape hatch" from the React paradigm.[3] Its purpose is to allow a component to "step outside" of React and synchronize its state with an *external system*. This external system could be a network API, the browser DOM, a setTimeout timer, or a third-party widget.

The core question of useEffect is not "When should this code run?" (e.g., "on mount"). The question is "What external system should this component's state be synchronized with?"

This distinction is critical because it clarifies when an Effect is *not* needed. There are two common cases where useEffect is frequently overused.

1. **Transforming data for rendering:** If data can be calculated purely from the current props or state, an Effect is not necessary. For example, filtering a list. A developer might be "tempted to write an Effect that updates a state variable when the list changes". This is inefficient, as it causes unnecessary render passes: the component renders once, the Effect runs, it updates state, and the component renders *again*. This logic should simply be placed at the top level of the component's render logic.
2. **Handling user events:** Logic that runs in response to a specific user action (like a button click) belongs in an event handler, not an Effect.[3]

The overuse of useEffect for state transformations is a direct symptom of the flawed "lifecycle" mental model. A developer thinks, "I need to react to a state change," which sounds like componentDidUpdate, and they reach for useEffect as its replacement. By reframing Effects as tools for *synchronization with external systems*, these anti-patterns can be avoided.

1.2. Deconstructing useEffect: Setup, Cleanup, and the Synchronization Cycle

The useEffect Hook is imported from React and called at the top level of a component. Its signature is useEffect(setup, dependencies?).

- **setup**: This is a function that contains the Effect's logic (e.g., connecting to a chat server).[5] React runs this setup function *after* the component has rendered and the changes have been committed to the DOM.
- **dependencies**: This is an optional array of "reactive values" (props, state, or other variables) used inside the setup function.[5]

A key feature of modern React development is React.StrictMode. When Strict Mode is on, React will intentionally run one extra development-only setup + cleanup cycle for every Effect on its first mount. This is not a bug; it is a deliberate pedagogical tool. The React team describes this as a "stress-test" to ensure that the cleanup logic "mirrors" the setup logic. If an Effect, for example, creates a subscription in setup but fails to properly remove it in cleanup, this double-run will immediately expose the resulting memory leak or bug. It forces the developer to design resilient Effects that can be started, stopped, and started again without issue, reinforcing the synchronization model.

1.3. Mastering the Dependency Array: A Strategic Guide

The dependency array is the most critical part of useEffect. It tells React *which* reactive values the Effect should "watch." The Effect's setup function will re-run only if one of these values has changed since the last render.

A common and dangerous anti-pattern is to "lie to the linter." The React linter provides a rule that checks all code inside useEffect and warns if a reactive value is used but not included in the dependency array. Developers, often wanting to achieve the "run once on mount" behavior of ``, will use an empty array even when the Effect *does* use props or state.

This leads to bugs. Using an empty array in this scenario "will definitely introduce bug[s] and [make the code] hard to maintain".[9] The Effect will capture a "stale" value of the prop or state from the initial render and will never "see" the updated values on subsequent renders. The intermediate lesson is to *trust the linter*. An empty `` array should be a declaration that "this Effect *truly* has no reactive dependencies," not a command to "only run this once."

1.4. The Critical Role of the Cleanup Function: Preventing Memory Leaks

The setup function passed to useEffect can optionally return another function. This is the **cleanup function**. Its primary purpose is to "stop synchronizing" and to prevent memory leaks by disposing of any resources the Effect created.

The name "cleanup" can be misleading. It does not *only* run when the component unmounts (the componentWillUnmount comparison).[1] The complete behavior, as specified in the React documentation, is:

1. After the initial render, React runs the setup function.
2. On a subsequent render, if any dependencies have changed:
   - React *first* runs the cleanup function from the *previous* render (with the old props/state).

    - React *then* runs the setup function from the *current* render (with the new props/state).
3. When the component unmounts, React runs the cleanup function from the final render.

The cleanup function is not just for unmounting; it is an integral part of the *update* and synchronization cycle.

Common scenarios requiring a cleanup function include:

- **Subscriptions:** If setup subscribes to a WebSocket or a browser event listener (e.g., window.addEventListener), the cleanup function must unsubscribe (e.g., window.removeEventListener).
- **Timers:** If setup creates a setTimeout or setInterval, cleanup must call clearTimeout or clearInterval.
- **API Requests:** To prevent a "state update on an unmounted component" warning, long-running fetch requests can be canceled. This is often done using an AbortController.
- **Async State:** A simple boolean flag can be used. The setup function creates a flag let unmounted = false;, and the cleanup function sets unmounted = true;. The async callback then checks if (!unmounted) before setting state.

Section 2: The useRef Hook: An Escape Hatch for References and Mutable Values

The useRef Hook is another "escape hatch" from the React paradigm. Unlike useState, which manages data used for rendering, useRef provides a way to hold onto a value that is *not* needed for rendering. It has two distinct primary use cases: accessing DOM elements and persisting mutable values.

2.1. Use Case 1: Accessing and Manipulating the DOM

The most common use case for useRef is to get a direct reference to a DOM element. This is necessary when a component needs to interact with non-React systems, such as the built-in browser APIs. Examples include managing focus, triggering animations, or integrating with third-party libraries.

The implementation follows three steps:

1. **Declare:** Import the hook and declare a ref, initializing it to null. const inputRef = useRef(null);.
2. **Attach:** Pass the ref object as the ref attribute in the JSX of the desired DOM node: <input ref={inputRef} />.
3. **Access:** After React creates the DOM node and renders it, it will automatically set the current property of the inputRef object to be that DOM node. This DOM node can then be accessed from event handlers or Effects.

The canonical example is programmatically focusing an input:

JavaScript

```javascript
import { useRef } from 'react';

export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    // Access the DOM node via.current
    // and call a browser API on it.
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputSef} />
      <button onClick={handleClick}>Focus the input</button>
    </>
  );
}
```

In this code, inputRef.current holds the actual <input> DOM element, allowing the handleClick function to call the .focus() method on it.

2.2. Use Case 2: Persisting Mutable Values Without Re-renders

The useRef Hook returns a plain JavaScript object with a single property called current. This object itself is persistent for the entire lifetime of the component, just like state.

The key behavior is that useRef *is not state*. Updating the current property of a ref *does not trigger a component re-render*. This makes useRef the ideal tool for storing information that needs to be persisted across renders but should not, by itself, cause a new render.

A common example is storing a setInterval ID:

JavaScript

```javascript
import { useState, useRef } from 'react';

function Stopwatch() {
  const = useState(0);
  const intervalRef = useRef(null);

  function handleStart() {
    intervalRef.current = setInterval(() => {
      setTime(prevTime => prevTime + 1);
    }, 1000);
  }

  function handleStop() {
    clearInterval(intervalRef.current);
  }

  //...
}
```

Here, intervalRef is used to store the ID returned by setInterval. This ID is needed by handleStop to clear the interval. If useState were used to store the ID, it would be unnecessary, as the ID value is never displayed in the UI.

This behavior has significant performance implications. Because useRef bypasses the render cycle, it is the correct tool for high-frequency updates that should not re-render the entire component. One anecdote highlights a 3D mouse-move effect: using useState "heated GPU," while useRef had "no heat at all," as it avoided re-rendering the component on every single pixel of mouse movement.

2.3. A Comparative Analysis: useRef vs. useState

The choice between useRef and useState is a fundamental architectural decision. The deciding question is: **"Is this information needed for rendering?"**.

- If the answer is **yes**, use useState. The data is part of the component's visible output, and changes to it *should* trigger a re-render.
- If the answer is **no**, and the value simply needs to be stored, use useRef.

Feature	**useState**	**useRef (Mutable Value)**
Primary Use	Storing data *used for rendering.*	Storing data *not* used for rendering.
Updating Value	setState(newValue)	ref.current = newValue
Triggers Re-render?	**Yes**	**No**
Value in Render	Acts like a *snapshot* for a given render.	Is *mutable*; ref.current can change during render.
Update Timing	*Asynchronous* (Update is scheduled).	*Synchronous* (Value changes immediately).
Common Example	const [count, setCount] =	const intervalRef = useRef(null)

	useState(0)	

A final caveat: while ref.current *can* be read during rendering, it should not be *written* to during rendering. Modifying a ref during render is a side effect that makes the component's behavior unpredictable. Ref mutations should occur inside event handlers or useEffect.

Section 3: Advanced Data Fetching Strategies

Data fetching is a form of side effect and a core task in most React applications. This section explores the tools for making HTTP requests and the patterns for managing the asynchronous state that results from them.

3.1. Native fetch API vs. axios: A Comparative Deep-Dive

React developers have two primary choices for data fetching: the built-in fetch API and the third-party axios library. The choice is not about *capability*—as fetch can be configured to do anything axios can —but about *developer experience* and the level of abstraction desired.

The fetch API is a modern, promise-based, native browser API. axios is a "batteries-included" library that simplifies many common networking tasks.

The core differences lie in the manual work fetch requires, which axios automates:

1. **JSON Parsing:** fetch resolves with a Response object. To get the JSON data, an explicit second step, response.json(), is required. axios automatically parses JSON, making the data available in response.data.
2. **Request Body:** When sending a POST request with fetch, the data must be manually stringified: body: JSON.stringify(data). axios handles this automatically.
3. **Error Handling:** This is the most significant difference. The fetch promise **does not reject** on HTTP errors like 404 (Not Found) or 500 (Server Error). It only rejects on network failures. The developer must manually check for response.ok or response.status to handle these errors. axios, by contrast, *does* reject the promise for any 4xx or 5xx status, which allows a .catch() block to handle all errors in one place.
4. **Interceptors:** axios provides "interceptors," a powerful feature for globally managing

requests and responses. This is commonly used to automatically attach authentication tokens to all outgoing requests or to handle global error responses. This logic must be built manually when using fetch.

Many developers find axios "easier" to learn and use precisely because it abstracts away the common "frustrations" of the fetch API.

3.2. Implementing Robust Data Flow: Loading States and Error Handling

Data fetching is asynchronous, meaning the component needs to manage *at least* three distinct states:

1. **Loading:** The request has been sent, but no response has been received.
2. **Success:** The request completed, and data is available.
3. **Error:** The request failed.

The standard pattern for managing this in a component involves three useState hooks : data, loading, and error.

JavaScript

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';

const MyComponent = () => {
  const  = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true); // Set loading state just before the request
      try {
        const { data: response } = await axios.get('/api/stuff');
        setData(response);
      } catch (error) {
        console.error(error.message);
        setError(error);
      }
      setLoading(false); // Set loading to false in all cases (success or error)
    };

    fetchData();
  },); // Empty array to run once on mount

  // UI logic renders based on the current state
  if (loading) {
    return <div>Loading...</div>; // [29, 30]
  }

  if (error) {
    return <div>Error: {error.message}</div>; // [30]
  }

  return (
    <div>
      {/* Render the successfully fetched data */}
      {data && <div>{data.name}</div>}
    </div>
  );
};
```

This pattern is robust, but it is also highly repetitive. Any component that needs to fetch data will duplicate this
useEffect and the three useState declarations. This repetition is the primary motivation for abstracting this logic into a custom useFetch hook, which is covered in Section 7.

3.3. Data Fetching in Effects: Best Practices and Common Pitfalls

The method shown above—fetching data inside a useEffect hook—is the "classic" or foundational way to handle data fetching in functional components. However, the React community and core team have increasingly moved away from this pattern.

A shift in best practices has emerged due to the limitations of useEffect for data fetching:

- **Effects don't run on the server:** This makes server-side rendering (SSR) with useEffect data fetching difficult or impossible.
- **They create "network waterfalls":** A parent component must render, then its useEffect fetches data. If a child component also needs to fetch data, it can only start *after* it renders, leading to a slow, sequential loading process.
- **They are "not very ergonomic":** The pattern lacks built-in features for caching, re-fetching on focus, or request de-duplication.

As of 2024, the "intermediate" and "advanced" best practice is to *avoid* using useEffect directly for data fetching. Instead, developers are strongly encouraged to use one of two modern solutions:

1. **Dedicated Data-Fetching Libraries:** Libraries like **React Query (TanStack)** or SWR have become the industry standard. These tools are not just data-fetching libraries; they are *server-state synchronization* libraries. They provide hooks (e.g., useQuery) that abstract away the useEffect logic and provide crucial, built-in features like automatic caching, background re-fetching, error handling, and retry logic.
2. **React Server Components (RSC):** In frameworks that support them (like Next.js), Server Components allow data to be fetched *on the server before* the component is rendered, completely eliminating the need for client-side useEffect fetching and its associated loading states.

For an intermediate developer, understanding the useEffect fetching pattern is essential for understanding the low-level mechanics. However, graduating to a tool like React Query is the next logical step for building efficient, production-grade applications.

Section 4: Architecting Forms in React

React's approach to forms is built on a fundamental architectural decision: whether the "source of truth" for the form's data lives in React state or in the DOM itself. This leads to two distinct patterns: **Controlled Components** and **Uncontrolled Components**.

4.1. The Controlled vs. Uncontrolled Component Dichotomy

Controlled Components
In a controlled component, the form's data is handled by the React component's state. The flow is as follows:

1. A useState hook is used to store the input's value.
2. The input's value attribute is bound to that state variable (e.g., value={name}).
3. The input's onChange attribute is bound to a function that calls the state setter (e.g., onChange={e => setName(e.target.value)}).

With this pattern, the component's state is the *single source of truth*. The input's value cannot change unless the state changes. This gives React complete control over the form.

Uncontrolled Components
In an uncontrolled component, the form data is handled by the DOM itself, just as in traditional HTML. The source of truth is the DOM node.40 React is not "aware" of the input's value as it changes.

- **Implementation (Modern):** To set an initial value, the defaultValue attribute is used (e.g., defaultValue="Bob"). The value is then read *only* upon submission by using the native FormData API inside the onSubmit event handler.
- **Implementation (Legacy):** An older, and now discouraged, pattern was to attach a ref to every single input (ref={this.input}) and read the value on submit via this.input.current.value. This pattern is considered "incorrect" and adds unnecessary boilerplate. The FormData API is the preferred modern approach for uncontrolled components.

The choice between these patterns involves significant trade-offs in performance, flexibility, and complexity.

4.2. Advanced Form Validation Techniques (Client-Side)

Form validation ensures the data entered by the user is correct and complete before submission. The chosen validation technique is often dictated by the form architecture (controlled vs. uncontrolled).

1. **Built-in HTML Validation:** This method uses native HTML5 attributes like required, pattern (for regex), minLength, maxLength, min, and max. This is the simplest form of validation, works well with uncontrolled components, and provides browser-native UI for errors.
2. **On-Submit Validation:** Validation logic is run inside the handleSubmit function *after* the user clicks "submit". This is the most common pattern for uncontrolled components, as the data is only available at submission time.
3. **Inline Validation (On Change):** Validation logic runs on *every keystroke* (i.e., in the onChange handler). This provides the best user experience, giving immediate feedback. This pattern is a natural fit for **controlled components**, as the value is already in state and available for validation on every change.

Modern form libraries like **React Hook Form** or **Formik** have become popular because they often provide the best of both worlds: they use uncontrolled components for performance but add the necessary logic to enable efficient inline validation, abstracting away the complexity. They also integrate well with schema-based validators like **Yup** or **Zod**.

4.3. Secure and Effective Form Submission Handling

Regardless of whether a form is controlled or uncontrolled, the submission process should be handled semantically and robustly. The unifying, standard practice is to use the <form> element's onSubmit event handler.

JavaScript

```javascript
const handleSubmit = (e) => {
  // 1. Prevent the default browser action (a full-page reload)
  e.preventDefault();

  // 2. Get the form data
  // For Uncontrolled:
  const formData = new FormData(e.target);
  const data = Object.fromEntries(formData.entries());
  // For Controlled:
  // const data = { name, email,... }; (already available in state)

  // 3. Perform final validation

  // 4. Send the data to the server (e.g., via fetch or axios)
};

return (
  <form onSubmit={handleSubmit}>
    {/* Form inputs go here */}
    <button type="submit">Submit</button>
  </form>
);
```

Using the onSubmit handler on the <form> element (rather than an onClick on the button) is the correct, accessible approach. It ensures the form can be submitted by pressing the "Enter" key and works correctly with assistive technologies. The e.preventDefault() call is essential to prevent the browser's default behavior and allow the React component to handle the submission via AJAX.

Section 5: Declarative Routing with React Router v6

React Router is the standard library for handling client-side routing in React applications. Version 6 introduced significant changes, culminating in the "data router" APIs (v6.4+), which are now the recommended standard.

5.1. Initializing the Router: createBrowserRouter and RouterProvider

The modern, recommended way to set up React Router is to define the application's routes as a data structure using createBrowserRouter. This is a paradigm shift from the older, component-based <BrowserRouter>. This new API is the only one that supports React Router's modern data loading and mutation features.

The setup is typically done in the application's entry point, main.jsx:

JavaScript

```javascript
import * as React from 'react';
import * as ReactDOM from 'react-dom/client';
import {
  createBrowserRouter,
  RouterProvider,
} from 'react-router-dom';
import Root from './routes/root'; // A layout component
import ErrorPage from './error-page'; // An error boundary component

// 1. Define routes as an array of objects
const router = createBrowserRouter();

// 2. Render the RouterProvider
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

The RouterProvider component is responsible for rendering the application and providing the routing context to all components.

5.2. Core Navigation: Link, useNavigate, and Route Definitions

React Router provides two primary ways to navigate between routes, which represent a fundamental split between *declarative* and *imperative* programming:

1. **Declarative Navigation (<Link>):** The <Link> component is the primary tool for navigation. It is used to create user-clickable navigation elements. It is declarative ("*what* to do") and renders a semantically correct <a> tag with the appropriate href.

```
JavaScript
import { Link } from 'react-router-dom';

<nav>
  <Link to="/">Home</Link>
  <Link to="/about">About</Link>
</nav>
```

2. **Imperative Navigation (useNavigate):** The useNavigate hook provides a navigate function that allows for programmatic navigation. It is imperative ("*how* and *when* to do it") and is used inside event handlers or side effects, such as after a successful form submission or login.

```
JavaScript
import { useNavigate } from 'react-router-dom';

function LoginForm() {
  const navigate = useNavigate();

  async function handleSubmit(e) {
    e.preventDefault();
    await loginUser(...);
    navigate('/dashboard'); // Programmatic navigation
  }
  //...
}
```

The heuristic is simple: if navigation is the direct result of a user clicking a link, use <Link>. If it is a side effect of other application logic, use useNavigate.

5.3. Dynamic and Nested UI: useParams, useSearchParams, and the <Outlet>

React Router excels at handling complex UI layouts and dynamic URLs.

Dynamic Routes (useParams)
To create routes with dynamic segments (e.g., a user ID), a colon (:) is used in the path

definition.

- **Definition:** path: "users/:userId"
- **Consumption:** The component rendered by this route (e.g., <ProfilePage />) can access the value of this segment using the useParams hook.[53]

  ```
  JavaScript
  import { useParams } from 'react-router-dom';

  function ProfilePage() {
    let { userId } = useParams(); // For URL /users/123, userId will be "123"
    return <h1>Profile for User {userId}</h1>;
  }
  ```

Query Strings (useSearchParams)

To read and write the URL's query string (e.g., /search?q=react), the useSearchParams hook is used. It works similarly to useState, returning a URLSearchParams object and a setter function.

- **Consumption:** let = useSearchParams();
- **Reading:** const query = searchParams.get('q');
- **Writing:** setSearchParams({ q: 'new-query' }); (This will update the URL).

Nested Routes (<Outlet>)

Nested routing is the key to creating compositional UI, where a "layout" component (like a sidebar) persists while the main content area changes.

- **Definition:** In createBrowserRouter, routes are nested by placing them in a children array.

  ```
  JavaScript
  const router = createBrowserRouter(,
    },
  ]);
  ```

- **Mechanism:** The parent route's element (<AppLayout />) must render an <Outlet /> component. This component acts as a *placeholder*, telling React Router where to render the matched child route's element (<Dashboard /> or <Profile />).

5.4. Advanced Patterns: Protected Routes and Authentication Flows

A "protected route" is one that should only be accessible to authenticated users.

The "Simple" Outlet-Based Pattern
The most common introductory pattern involves creating a wrapper component (e.g., <ProtectedRoute>) that checks the user's authentication status.

- If the user is authenticated, the component renders an <Outlet />, which will render the intended child route.
- If the user is *not* authenticated, the component renders a <Navigate to="/login" replace /> component, redirecting them to the login page.

The "Intermediate" Problem: A Loader Race Condition
This simple pattern, while common, has a critical flaw when used with the modern data router: it creates a race condition with data loaders.
A developer's common complaint is this: "The issue... is that the *loaders* of any matching child route... are fired *before* the ProtectedRoute component is rendered and the Authentication is checked".

This means a user who is not logged in might navigate to /dashboard. The router will *first* fire the /dashboard loader (which might try to fetch sensitive data), that loader will fail (because the user is unauthenticated), and *then* the ProtectedRoute component will render and redirect the user to /login. The user may see a "401 Unauthorized" error flash on the screen *before* they are redirected.

This is a significant problem. An expert-level report must acknowledge that the simple <Outlet> pattern is now insufficient. The "intermediate" challenge is to solve authentication *at the loader level*. This involves having every protected loader *first* check the authentication status (e.g., from a context or auth service) *before* attempting its data fetch, redirecting from within the loader itself if necessary.

Section 6: Global State Management with the Context API

As applications grow, passing props from a top-level component down to a deeply nested child component becomes "verbose and inconvenient".[67] This problem is known as **"prop drilling"**.

6.1. Introduction to useContext: Escaping Prop Drilling

The React Context API provides a solution to prop drilling. It creates a way to "teleport" data from a parent "provider" component to *any* component in the tree below it, "no matter how deep," without passing it through props at every intermediate level.

Context is specifically designed to share data that can be considered "global" for a tree of components, such as the current authenticated user, the UI theme (e.g., "dark" or "light"), or a preferred language.

However, Context is not a silver bullet for all state management. It comes with trade-offs.

- **Debugging:** It can be "more difficult to debug" than props, as it is less explicit where the data is coming from.
- **Performance:** When a Context Provider's value changes, *all* components consuming that context will re-render. This makes Context a poor choice for high-frequency state, such as the value of a form input. Using Context for such state "is actually worse as it'll rerender all the consumers".

For complex, high-frequency global state, intermediate developers should consider dedicated state management libraries like Zustand or Redux, which offer more granular control over re-renders. Context is best suited for *low-frequency* global data.

6.2. Implementing Context: createContext, Provider, and Consumer Patterns

Implementing Context involves three steps:

1. **Create:** First, a new context is created using createContext. This is typically done in its own file to be easily imported. The function accepts a defaultValue, which is the value a consumer will receive *only if* it is rendered without a matching Provider in the tree above it.
   JavaScript
   ```javascript
   // ThemeContext.js
   import { createContext } from 'react';
   export const ThemeContext = createContext('light'); // 'light' is the default value
   ```

2. **Provide:** In a parent component, import the context and use its Provider component. This

component must be wrapped around the part of the component tree that needs access to the data. The data is passed via the value prop.

```javascript
// App.js
import { ThemeContext } from './ThemeContext';
import { useState } from 'react';

function App() {
  const = useState('dark');

  return (
    <ThemeContext.Provider value={theme}>
      {/* All children can now access the 'dark' value */}
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

3. **Consume (Modern):** In any child component, no matter how deep, the value is read using the useContext hook.

```javascript
// Button.js
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

function Button() {
  const theme = useContext(ThemeContext); // theme will be 'dark'
  return <button className={theme}>Click Me</button>;
}
```

Legacy Consumer Pattern: Before hooks, Context was consumed using a "render prop" component (<MyContext.Consumer>). This pattern is "rarely used" today but may be seen in older codebases or used in class components.

Section 7: Encapsulating Logic: Creating Custom Hooks

Custom Hooks are the primary pattern for logic sharing in modern React. They are a powerful feature that allows developers to "extract component logic into reusable functions".

7.1. The Rules, Naming Conventions, and Design of Hooks

A custom hook is, fundamentally, just a JavaScript function with two special properties:

1. Its name **must** start with the prefix use (e.g., useFetch, useLocalStorage).
2. It can call other hooks (like useState, useEffect, or useContext).

The use prefix is not just a convention; it is a mandatory rule that allows React's linter to enforce the "Rules of Hooks" (e.g., "don't call hooks in loops") inside the custom hook.

Custom hooks are the modern replacement for older logic-sharing patterns like Higher-Order Components (HOCs) and Render Props.[81] They solve the same problem—reusing stateful logic—without the "wrapper hell" or complex component composition that those patterns required.

Best Practices for Design:

- **Descriptive Names:** The name should clearly reflect the hook's purpose. useChatMessages is better than useMessages.
- **Single Focus:** A hook should encapsulate a single piece of functionality (e.g., data fetching, or local storage, but not both).
- **Return Necessary Data:** The hook should return only what the consuming component needs, typically an object or an array.

7.2. Practical Example 1: Building a Reusable useFetch Hook

As identified in Section 3.2, the logic for managing data, loading, and error states is highly repetitive. A useFetch hook is the perfect solution to abstract this logic.

JavaScript

```
import { useState, useEffect } from 'react';

const useFetch = (url) => {
  const = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    // AbortController is used to cancel the fetch on cleanup
    const controller = new AbortController();

    const fetchData = async () => {
      setLoading(true); // Ensure loading is true on new URL
      try {
        const response = await fetch(url, { signal: controller.signal });
        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }
        const json = await response.json();
        setData(json);
        setError(null); // Clear previous errors
      } catch (e) {
        if (e.name!== 'AbortError') {
          setError(e);
        }
      } finally {
        setLoading(false);
      }
    };

    fetchData();

    // Cleanup function: abort the fetch if the component unmounts or URL changes
    return () => {
      controller.abort();
    };
  }, [url]); // Re-run the effect if the URL prop changes
  return { data, loading, error };
};
```

A component can now consume this complex logic in one line:
const { data, loading, error } = useFetch('https://api.example.com/users');.

7.3. Practical Example 2: Building a useLocalStorage Hook

Another common use case is synchronizing a component's state with the browser's localStorage API. This hook provides a useState-like interface but with persistence.

JavaScript

```javascript
import { useState } from 'react';

export const useLocalStorage = (key, defaultValue) => {
  // 1. Get initial value from localStorage or use default
  const getInitialValue = () => {
    try {
      const item = window.localStorage.getItem(key); // [85]
      return item? JSON.parse(item) : defaultValue;
    } catch (error) {
      console.log(error);
      return defaultValue;
    }
  };

  // 2. Use useState, initializing it with our new function
  const = useState(getInitialValue);

  // 3. Create a new "setter" function
  const setValue = (value) => {
    try {
      // Allow value to be a function (like useState's setter)
      const valueToStore =
        value instanceof Function? value(storedValue) : value;

      // 4. Update state
      setStoredValue(valueToStore);

      // 5. Update localStorage
      window.localStorage.setItem(key, JSON.stringify(valueToStore)); // [85]
    } catch (error) {
      console.log(error);
    }
  };

  // 6. Return the useState-like tuple
  return [storedValue, setValue];
};
```

This hook can now be used to create, for example, a persistent form input or a "remember me" checkbox, as seen in the drawing app example.

7.4. Best Practices for Designing Maintainable Custom Hooks

Custom hooks are the ultimate expression of the useEffect philosophy. Section 1 established that Effects are for *synchronizing* React state with an *external system.*

Custom hooks are the design pattern for packaging these synchronization units into reusable, descriptive, and maintainable abstractions:

- useFetch synchronizes component state with a **remote network API**.
- useLocalStorage synchronizes component state with the **browser's storage API**.
- useOnlineStatus (another common example) synchronizes component state with the **browser's network status API**.

useEffect is the low-level tool; useCustomHook is the high-level, abstract pattern that makes this synchronization philosophy practical and scalable. Mastering this pattern is the key to moving from an intermediate to an advanced React developer.